PATENT ABSTRACTS OF JAPAN

(11)Publication number : 2001-216161
(43)Date of publication of application : 10.08.2001

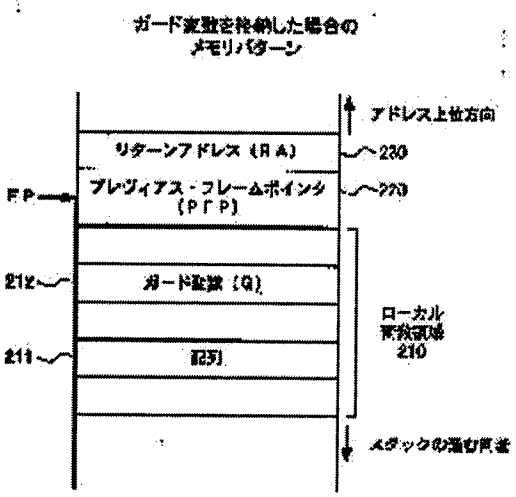
(51)Int.Cl. G06F 9/42

(21)Application number : 2000-028358
(22)Date of filing : 04.02.2000

(71)Applicant : INTERNATL BUSINESS MACH CORP <IBM>
(72)Inventor : ETO HIROAKI
YODA KUNIKAZU

(54) MEMORY DEVICE, STACK PROTECTION SYSTEM, COMPUTER SYSTEM, COMPILER, STACK PROTECTING METHOD, STORAGE MEDIUM AND PROGRAM TRANSMITTER

(57)Abstract:
PROBLEM TO BE SOLVED: To prevent stack smashing attack due to buffer overflow on a stack.
SOLUTION: This memory device is used by a computer system. It is provided with an area to store a return address 230 to a call origin of a function under execution, an area to store a previous frame pointer 22 to the call origin of the function under execution and an area to be located in the rear of the area to store the return address 230 and the area to store the previous frame pointer 220 and to store local variables as memory patterns of this memory device after invoking the function when this computer system executes programs. The area to store the local variables stores a guard variable 212 as an object to confirm whether or not it is destroyed in a return processing of the function under execution before this arrangement 211 when the arrangement 211 is stored in the area to store the local variables.



LEGAL STATUS

[Date of request for examination] 29.09.2000
[Date of sending the examiner's decision of rejection]
[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]
[Date of final disposal for application]
[Patent number] 3552627
[Date of registration] 14.05.2004
[Number of appeal against examiner's decision of rejection]
[Date of requesting appeal against examiner's decision of rejection]
[Date of extinction of right]

(19)日本国特許庁（ＪＰ）　(12)**公開特許公報（Ａ）**　(11)特許出願公開番号
特開2001－216161
（P2001－216161A）
(43)公開日　平成13年８月10日(2001.8.10)

(51)Int.Cl.⁷	識別記号	ＦＩ	ﾃｰﾏｺｰﾄﾞ(参考)
Ｇ０６Ｆ　9/42	３３０	Ｇ０６Ｆ　9/42	３３０Ｃ　５Ｂ０３３

審査請求　有　　請求項の数19　ＯＬ　（全 15 頁）

(21)出願番号　特願2000－28358(P2000－28358)

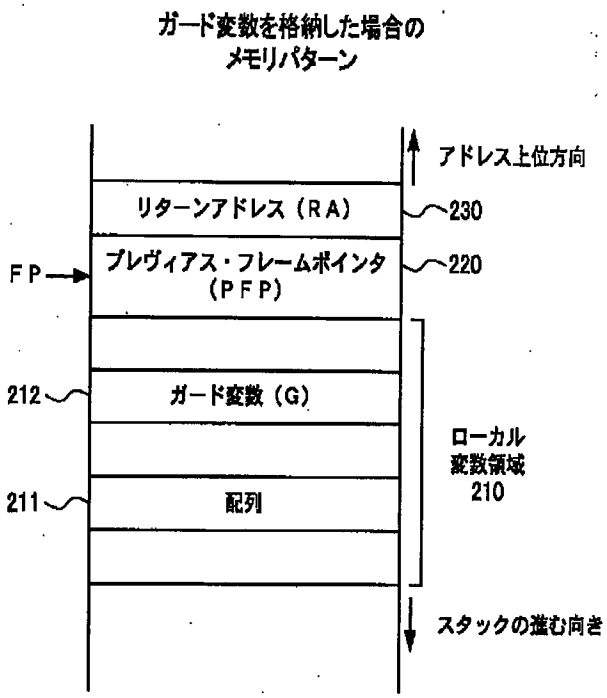
(22)出願日　平成12年２月４日(2000.2.4)

(71)出願人　390009531
インターナショナル・ビジネス・マシーンズ・コーポレーション
ＩＮＴＥＲＮＡＴＩＯＮＡＬ　ＢＵＳＩＮＥＳＳ　ＭＡＳＣＨＩＮＥＳ　ＣＯＲＰＯＲＡＴＩＯＮ
アメリカ合衆国10504、ニューヨーク州　アーモンク　（番地なし）
(74)復代理人　100104880
弁理士　古部　次郎　（外３名）

最終頁に続く

(54)【発明の名称】　メモリ装置、スタック保護システム、コンピュータシステム、コンパイラ、スタック保護方法、記憶媒体及びプログラム伝送装置

(57)【要約】
【課題】　スタック上のバッファオーバーフローを原因とするスタックスマッシング攻撃を防止する。
【解決手段】　コンピュータシステムが使用するメモリ装置であって、このコンピュータシステムがプログラムを実行する際の関数呼び出し後におけるこのメモリ装置のメモリパターンとして、実行中の関数の呼び出し元へのリターンアドレス２３０を格納する領域と、この実行中の関数の呼び出し元へのプレヴィアス・フレームポインタ２２を格納する領域と、このリターンアドレス２３０を格納する領域とこのプレヴィアス・フレームポインタ２２０を格納する領域との後方に位置し、ローカル変数を格納する領域とを備え、このローカル変数を格納する領域は、このローカル変数を格納する領域に配列２１１が格納されている場合に、この配列２１１よりも前に、実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数２１２を格納する。



【特許請求の範囲】
【請求項１】　コンピュータシステムが使用するメモリ装置であって、
前記コンピュータシステムがプログラムを実行する際の関数呼び出し後における前記メモリ装置のメモリパターンとして、
実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、
前記実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、
前記リターンアドレスを格納する領域と前記プレヴィアス・フレームポインタを格納する領域との後方に位置し、ローカル変数を格納する領域とを備え、
前記ローカル変数を格納する領域は、当該ローカル変数を格納する領域に配列が格納されている場合に、当該配列よりも前に、前記実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したメモリパターンを有することを特徴とするメモリ装置。
【請求項２】　前記メモリ装置の前記メモリパターンにおける前記ローカル変数を格納する領域は、当該ローカル変数を格納する領域に文字配列が格納されている場合に、当該配列よりも前に、前記ガード変数を格納したことを特徴とする請求項１に記載のメモリ装置。
【請求項３】　前記メモリ装置の前記メモリパターンにおける前記ローカル変数を格納する領域に格納される前記ガード変数は乱数であることを特徴とする請求項１に記載のメモリ装置。
【請求項４】　コンピュータシステムが使用するメモリ装置であって、
前記コンピュータシステムがプログラムを実行する際の関数呼び出し後における前記メモリ装置のメモリパターンとして、
実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、
前記リターンアドレスを格納する領域の後方に位置し、前記実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、
前記プレヴィアス・フレームポインタを格納する領域の後方に位置し、ローカル変数を格納する領域とを備え、
前記ローカル変数を格納する領域に配列が格納されている場合に、前記プレヴィアス・フレームポインタと当該配列との間に、前記実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したメモリパターンを有することを特徴とするメモリ装置。
【請求項５】　コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護システムであって、
ソース形式のプログラムを入力し、関数呼び出し後のスタックにおけるプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される配列との間にガード変数を格納する指令を当該ソース形式のプログラムに付加するスタック保護指令作成部と、
前記スタック保護指令作成部により前記ガード変数を格納する指令を付加されたプログラムを実行し、当該ガード変数を格納する指令にしたがって、関数呼び出しの際にスタックに当該ガード変数を格納すると共に、当該関数のリターン処理において当該ガード変数の有効性を確認するスタック保護処理実行部とを備えることを特徴とするスタック保護システム。
【請求項６】　前記スタック保護処理実行部は、
前記関数のリターン処理において前記ガード変数が破壊されていることを検出した場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行うことを特徴とする請求項５に記載のスタック保護システム。
【請求項７】　前記スタック保護指令作成部は、コンパイラに実装され、前記コンパイラが、前記ソース形式のプログラムとしてコンパイラ言語にて記述されたソースプログラムを入力し、当該ソースプログラムをオブジェクトプログラムに翻訳する際に、当該オブジェクトプログラムに対して、前記ガード変数を格納する指令を付加することを特徴とする請求項５に記載のスタック保護システム。
【請求項８】　コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護システムであって、
前記関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令が付加されたプログラムを入力して実行すると共に、当該ガード変数を格納する指令にしたがって、関数呼び出しの際にスタックに前記ガード変数を格納し、当該関数のリターン処理において当該ガード変数の有効性を確認するプログラム実行手段と、
前記プログラム実行手段により、前記関数のリターン処理において前記ガード変数が破壊されていることが検出された場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行う異常終了実行手段とを備えることを特徴とするスタック保護システム。
【請求項９】　種々の演算処理を行うデータ処理装置と、当該データ処理装置が演算処理を行う際に用いるメモリ装置とを備えたコンピュータシステムであって、
前記データ処理装置が演算処理を実行する際の関数呼び出し後における前記メモリ装置のメモリパターンは、
実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、
前記実行中の関数の呼び出し元へのプレヴィアス・フレ

ームポインタを格納する領域と、
前記リターンアドレスを格納する領域と前記プレヴィアス・フレームポインタを格納する領域との後方に位置し、ローカル変数を格納する領域とを備え、
前記ローカル変数を格納する領域は、当該ローカル変数を格納する領域に配列が格納されている場合に、当該配列よりも前に、前記実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したことを特徴とするコンピュータシステム。

【請求項１０】　種々の演算処理を行うデータ処理装置と、当該データ処理装置が演算処理を行う際に用いるメモリ装置とを備えたコンピュータシステムであって、
前記データ処理装置が演算処理を実行する際の関数呼び出し後における前記メモリ装置のメモリパターンは、
実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、
前記リターンアドレスを格納する領域の後方に位置し、前記実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、
前記プレヴィアス・フレームポインタを格納する領域の後方に位置し、ローカル変数を格納する領域とを備え、
前記ローカル変数を格納する領域に配列が格納されている場合に、前記プレヴィアス・フレームポインタと当該配列との間に、前記実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したことを特徴とするコンピュータシステム。

【請求項１１】　プログラム制御により種々の演算処理を行うコンピュータシステムであって、
関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令が付加されたプログラムを読み込んで実行するデータ処理装置と、
前記データ処理装置が演算処理を行う際に用いるメモリ装置とを備え、
前記データ処理装置は、前記ガード変数を格納する指令が付加されたプログラムにおける前記ガード変数を格納する指令にしたがって、関数呼び出しの際に、前記メモリ装置上のスタックに当該ガード変数を格納すると共に、当該関数のリターン処理において当該ガード変数の有効性を確認することを特徴とするコンピュータシステム。

【請求項１２】　前記データ処理装置は、
前記関数のリターン処理において前記ガード変数が破壊されていることを検出した場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行うことを特徴とする請求項１１に記載のコンピュータシステム。

【請求項１３】　プログラム制御により種々の演算処理を行うコンピュータシステムであって、
ソース形式のプログラムを入力し、関数呼び出し後のスタックにおけるプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される配列との間にガード変数を格納する指令を当該ソース形式のプログラムに付加すると共に、当該ガード変数を格納する指令を付加されたプログラムを実行するデータ処理装置と、
前記データ処理装置が演算処理を行う際に用いるメモリ装置とを備え、
前記データ処理装置は、前記ガード変数を格納する指令を付加されたプログラムにおける前記ガード変数を格納する指令にしたがって、関数呼び出しの際に、前記メモリ装置上のスタックに当該ガード変数を格納すると共に、当該関数のリターン処理において当該ガード変数の有効性を確認することを特徴とするコンピュータシステム。

【請求項１４】　前記データ処理装置は、
前記関数のリターン処理において前記ガード変数が破壊されていることを検出した場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行うことを特徴とする請求項１３に記載のコンピュータシステム。

【請求項１５】　ソースプログラムを入力しオブジェクトプログラムに翻訳して出力するコンパイラであって、
プログラムを翻訳する翻訳手段と、
プログラムの翻訳の際に、サブルーチンごとに、当該サブルーチンに含まれる関数が配列を持っているかどうかを調べ、配列を持っている場合に、当該サブルーチンに対して、当該関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される当該配列との間にガード変数を格納すると共に、関数のリターン処理において当該ガード変数の有効性を確認する旨の指令を付加するスタック保護指令付加手段とを備えることを特徴とするコンパイラ。

【請求項１６】　コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護方法であって、
ソース形式のプログラムに対して、関数呼び出し後のスタックにおけるプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される配列との間にガード変数を格納する指令を当該ソース形式のプログラムに付加するステップと、
前記ガード変数を格納する指令を付加されたプログラムを実行すると共に、当該ガード変数を格納する指令にしたがって、関数呼び出しの際にスタックに当該ガード変数を格納し、関数のリターン処理において当該ガード変数の有効性を確認するステップと、
前記関数のリターン処理において前記ガード変数が破壊

されていることが検出された場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知するステップとを含むことを特徴とするスタック保護方法。

【請求項１７】　コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護方法であって、

前記関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令が付加されたプログラムを実行すると共に、当該ガード変数を格納する指令にしたがって、関数呼び出しの際にスタックに当該ガード変数を格納し、関数のリターン処理において当該ガード変数の有効性を確認するステップと、

前記関数のリターン処理において前記ガード変数が破壊されていることが検出された場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知するステップとを含むことを特徴とするスタック保護方法。

【請求項１８】　コンピュータに実行させるプログラムを当該コンピュータの入力手段が読取可能に記憶した記憶媒体において、

前記プログラムは、

配列を持っている関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される当該配列との間にガード変数を格納する処理と、

前記関数のリターン処理において前記ガード変数の有効性を確認する処理と、

前記関数のリターン処理において前記ガード変数が破壊されていることが検出された場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する処理とを前記コンピュータに実行させることを特徴とする記憶媒体。

【請求項１９】　コンピュータに、

配列を持っている関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数として当該スタックに格納される当該配列との間にガード変数を格納する処理と、前記関数のリターン処理において当該ガード変数の有効性を確認する処理と、前記関数のリターン処理において前記ガード変数が破壊されていることが検出された場合に、前記プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する処理とを実行させるプログラムを記憶する記憶手段と、

前記記憶手段から前記プログラムを読み出して当該プログラムを送信する送信手段とを備えたことを特徴とするプログラム伝送装置。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】本発明は、スタックスマッシング攻撃によるコンピュータへの侵入からプログラムカウンタを保護するスタック保護方法に関する。

【０００２】

【従来の技術】コンピュータシステムへの侵入は、一般に、システム設定の不備やセキュリティホールと呼ばれるシステムの弱点を攻撃して他人の権限で侵入し、次の攻撃でシステム管理者権限を得るといった手順で行われる。

【０００３】このような攻撃の典型的なものとしてスタックスマッシング攻撃がある。スタックには、プログラム内のサブルーチンの実行状態を保持するために、サブルーチンのリターンアドレス、ローカル変数、引数などが保管される。そして、スタックスマッシング攻撃は、スタック内のリターンアドレスを書き換えることによって悪意のあるプログラムに実行制御を移すという手法である。

【０００４】スタックの構造を示してスタックスマッシング攻撃について具体的に説明する。図９はＣ言語における関数呼び出し後のスタック状態を示している。図９を参照すると、スタックトップ（図では下側）から順に、ローカル変数（Local Variable：ＬＶ）、当該関数の呼び出し元である直前の関数のフレームポインタ（Previous frame pointer：ＰＦＰ）、及び当該呼び出し元へのリターンアドレス（Return Address：ＲＡ）が積まれている。ここで、フレームポインタ（frame pointer：ＦＰ）とは、ローカル変数が相対参照される際のベースポインタの事である。スタックスマッシング攻撃においては、ローカル変数（特に配列）への代入操作を行い、境界チェックが不完全な場合に、そのローカル変数より上位のスタック領域を破壊する。そして、スタック領域を破壊する際に、プログラムを書き込むと共に、リターンアドレスの領域にそのプログラムを指すような値を書き込む。この状態で関数のリターン処理が行われると、書き込まれたプログラムが起動することとなる。

【０００５】Ｃ言語で記述されたプログラムを例にして、スタックスマッシング攻撃について、さらに具体的に説明する。図１０はスタックスマッシング攻撃の攻撃対象であるＣ言語で記述されたプログラムの例であり、図１１はこのプログラムに対してスタックスマッシング攻撃が行われた場合のスタックの状態を説明する図である。図１０において、関数fooは環境変数HOMEの内容をローカル変数にコピーするプログラムである。関数strcpyはローカル変数の大きさを考慮しないため、環境変数HOMEの内容が１２８文字以上である場合、図１１のbufから上方向の内容(網掛け箇所)を壊すことになる。この破壊内容にプログラムを埋め込み、かつリターンアドレスの領域にそのプログラムの先頭アドレスを置くことによって、埋め込んだプログラムへ制御を移すことができる。

【０００６】上述したスタックスマッシング攻撃などにより、他人の管理するコンピュータネットワークに侵入する行為は、それ自体が機密漏洩やシステム破壊などを引き起こす場合がある。また、他人のＩＤやパスワードを盗み、その正当な所有者のふりをしてシステムに侵入するなりすまし行為や、他のコンピュータに侵入するための足がかりとして利用される場合もある。従来、この種のスタックスマッシング攻撃は、ＵＮＩＸサーバを主な攻撃目標としていた。しかし最近では、ActiveX（Acrobat Control for ActiveXなど（なお、ActiveXは米国マイクロソフト・コーポレーションの商標）のバッファオーバーフローが問題となり、これを対象とする攻撃が行われるようになっている。そのため、ネットワークに接続するコンピューター般、すなわち、サーバマシン、クライアントマシン、ＰＤＡなどの全てが攻撃目標となり得る。バッファオーバーフローを起こすバグは、発見されてから取り除かれるというのが一般的である。したがって、このバグに対する攻撃を防ぐには、そのような攻撃に対する何らかの予防措置をとることが有効であると考えられる。そこで、これらのコンピュータシステムへの侵入を防止するために、従来から種々の方策が提案されている。

【０００７】この種の従来技術の例としては、例えば、文献「Automatic Detection and Prevention of Buffer-Overflow Attacks」（Crispin Cowan, Calton Pu, David Majer, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, in the 7th USENIX Security Symposium, San Antonio, TX, January 1998.）に開示されたStackGuardという技術がある。StackGuardでは、図９に示したようなスタック状態におけるリターンアドレスの領域と呼び出し元のフレームポインタの領域との間に、リターンアドレスの保護のための特別の値であるguard_valueを挿入する。そして、関数のリターン処理において、guard_valueの有効性を確認することにより、攻撃の有無を確認する。

【０００８】上記従来技術の他にも、仮想記憶のページを操作し、スタック上のリターンアドレス格納箇所を書き込み禁止にする手法や、リターンアドレス専用のスタックを別に作成し、関数の入口でリターンアドレスを専用スタックにコピーし、関数リターン処理で専用スタックから戻すことにより、リターンアドレスを保護する手法などが提案されている。また、バッファオーバーフローを原因とするあらゆる攻撃を防止するために、全ての配列アクセスに対して境界チェックを行うという手法も考えられる。

【０００９】

【発明が解決しようとする課題】しかし、上述したコンピュータシステムへの侵入を防止するための従来技術は、次のような欠点を有している。上記文献に記載されたStackGuardは、実行中の関数の呼び出し元である直前の関数へのリターンアドレスと当該直前の関数におけるフレームポインタ（以下、ＰＦＰと略す）との間に、guard_valueを格納する。このため、リターンアドレスは保護されているが、変数スコープを指し示すフレームポインタは保護されていない。これにより、重大なセキュリティホールが存在している。すなわち、フレームポインタを保護していないために、プログラムカウンタを操作することが可能である。以下に、その概要を説明する。

【００１０】関数リターン処理では、次の各処理が行われる。

ＦＰ　　→　　ＳＰ
ＰＦＰ　→　　ＦＰ
（ＳＰ）　→　　ＰＣ　　；return operation

ここで、ＦＰはフレームポインタ、ＳＰはスタックポインタ、ＰＣはプログラムカウンタである。また、矢印は値を代入することを示し、括弧は間接参照することを示す。上記のように、StackGuardでは、スタック内のＰＦＰを保護していない。そのため、外部からの攻撃により、ＰＦＰの値を操作することが可能である。値の操作について、その伝搬を調べると、上記の関数リターン処理を参照すれば、ＰＦＰを操作することにより一度のリターンでフレームポインタを操作することができる。そして、二度目のリターンでプログラムカウンタを操作することができる。実際に悪意のあるコードを埋め込むには、guard_valueテストを通過させなくてはならないが、StackGuardがデフォルトで使用しているguard_valueに対してはスタックスマッシング攻撃が可能である。そしてさらに、その他のguard_valueについても攻撃手法が見つかる可能性がある。したがって、StackGuardで用いられる従来の手法では、スタックスマッシング攻撃を完全に防止することができなかった。

【００１１】また、StackGuardでは、一度目のリターンで汚されたフレームポインタの影響を検出するために、二度目のリターンでは必ずguard_valueの有効性を確認しなくてはならない。そのため、全ての関数において、guard_valueの有効性を確認するプログラムを発生させている。したがって、guard_valueの有効性を確認する処理のために、必ずある程度のオーバーヘッドが発生していた。さらにまた、StackGuardにおいては、リターンアドレス領域とＰＦＰ領域との間にguard_valueを挿入するため、スタックのフレーム構造を変更することとなる。このため、StackGuardを実装したプログラムに対してデバッグ支援を行うことができなかった。

【００１２】仮想記憶のページを操作し、スタック上のリターンアドレス格納箇所を書き込み禁止にする従来技術においては、リターンアドレスのみを保護し、フレームポインタを保護していないため、StackGuardと同様のセキュリティホールが存在し、スタックスマッシング攻撃を完全に防止することができなかった。また、スタッ

クページへの書き込みを行うときにはスーパーバイザーモードに制御が移るため、実行オーバーヘッドが大きいという欠点があった。

【００１３】リターンアドレス専用のスタックを別に作成し、関数の入口でリターンアドレスを専用スタックにコピーし、関数リターン処理で専用スタックから戻す従来技術においては、リターンアドレスへの直接的な攻撃が可能であった。すなわち、グローバル変数領域に作成された専用スタックは、その値の有効性を確認しないで元のスタックに戻すため、専用スタックからリターンアドレスを戻す操作への攻撃が直接リターンアドレスへの攻撃となってしまう。また、同従来技術は、専用スタックの管理オーバーヘッドが大きいという欠点があった。

【００１４】全ての配列アクセスに対して境界チェックを行う場合は、上述したようにバッファオーバーフローを原因とするあらゆる攻撃を防止することができるが、主にポインタを使用したプログラムではオーバーヘッドが非常に大きいという欠点があった。

【００１５】本発明は以上のような技術的課題を解決するためになされたものであって、スタック上のバッファオーバーフローを原因とするスタックスマッシング攻撃を完全に防止することを目的とする。また、スタックスマッシング攻撃の防止手段を実装したことによる実行オーバーヘッドを削減することを他の目的とする。

【００１６】

【課題を解決するための手段】かかる目的のもと、本発明は、コンピュータシステムが使用するメモリ装置であって、このコンピュータシステムがプログラムを実行する際の関数呼び出し後におけるこのメモリ装置のメモリパターンとして、実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、この実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、このリターンアドレスを格納する領域とこのプレヴィアス・フレームポインタを格納する領域との後方に位置し、ローカル変数を格納する領域とを備え、このローカル変数を格納する領域は、このローカル変数を格納する領域に配列が格納されている場合に、この配列よりも前に、実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したメモリパターンを有することを特徴としている。このような構成とすれば、リターンアドレスと共にプレヴィアス・フレームポインタをも保護することにより、スタックスマッシング攻撃を用いたシステムへの侵入を確実に阻止することができる。さらに、ガード変数を用いたスタック保護処理の対象を、ローカル変数を格納する領域に配列が格納されている場合に限定することにより、システム全体のオーバーヘッドを低減させることができる。

【００１７】ここで、メモリ装置のメモリパターンにおけるローカル変数を格納する領域は、このローカル変数を格納する領域に文字配列が格納されている場合に、この配列よりも前に、ガード変数を格納する構成とすることができる。すなわち、ローカル変数を格納する領域に、配列のうちの文字配列が格納されている場合にのみ、ガード変数を用いたスタック保護処理を実行する。このような構成とすれば、文字配列を持たない関数の他に整数配列のような文字配列以外の配列を持つ関数に対してもガード変数を用いたスタック保護処理を行わないため、システム全体のオーバーヘッドを更に低減させることができる。スタックスマッシング攻撃の対象となる配列は、主に文字配列であるため、ガード変数を用いたスタック保護処理の対象を文字配列を持つ関数のみに限定しても、安全性が大幅に損なわれることはない。そこで、処理速度が重視されるシステムでは、このような構成とすることが好ましい。

【００１８】また、ローカル変数を格納する領域に格納されるガード変数として、乱数を用いる構成とすることができる。このほか、ガード変数としては、攻撃者が知ることのできない任意の値を用いることができる。

【００１９】さらに本発明においては、コンピュータシステムが使用するメモリ装置であって、このコンピュータシステムがプログラムを実行する際の関数呼び出し後におけるこのメモリ装置のメモリパターンとして、実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、このリターンアドレスを格納する領域の後方に位置し、実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、このプレヴィアス・フレームポインタを格納する領域の後方に位置し、ローカル変数を格納する領域とを備え、このローカル変数を格納する領域に配列が格納されている場合に、このプレヴィアス・フレームポインタとこの配列との間に、実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したメモリパターンを有することを特徴としている。すなわち、ガード変数を格納する位置は、スタック上のプレヴィアス・フレームポインタと配列との間であれば、どこでも良い。

【００２０】また、本発明は、コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護システムであって、ソース形式のプログラムを入力し、関数呼び出し後のスタックにおけるプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納する指令をこのソース形式のプログラムに付加するスタック保護指令作成部と、このスタック保護指令作成部によりガード変数を格納する指令を付加されたプログラムを実行し、このガード変数を格納する指令にしたがって、関数呼び出しの際にスタックにこのガード変数を格納すると共に、この関数のリターン処理においてこのガード変数の有効性を確認するスタッ

ク保護処理実行部とを備えることを特徴としている。このような構成とすれば、プログラムを実行する際に、リターンアドレスと共にプレヴィアス・フレームポインタをも保護することができ、スタックスマッシング攻撃を用いたシステムへの侵入を確実に阻止することができる点で優れている。

【００２１】ここで、スタック保護処理実行部は、関数のリターン処理においてガード変数が破壊されていることを検出した場合に、プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行う構成とすることができる。このような構成とすれば、スタックスマッシング攻撃が行われた場合には、かかる関数の呼び出し元へ処理が戻る直前でプログラムの実行を中止するため、システムへの侵入を効果的に防止することができる点で好ましい。

【００２２】また、スタック保護指令作成部は、コンパイラに実装され、このコンパイラが、ソース形式のプログラムとしてコンパイラ言語にて記述されたソースプログラムを入力し、このソースプログラムをオブジェクトプログラムに翻訳する際に、このオブジェクトプログラムに対して、ガード変数を格納する指令を付加する構成とすることができる。このような構成とすれば、オーバーフローを引き起こす変数よりもスタック上で上位アドレスにガード変数の宣言を配置すること、及び全ての関数の出口にプログラムを挿入することが容易となる点で好ましい。

【００２３】さらに本発明においては、コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護システムであって、関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令が付加されたプログラムを入力して実行すると共に、このガード変数を格納する指令にしたがって、関数呼び出しの際にスタックに前記ガード変数を格納し、この関数のリターン処理においてこのガード変数の有効性を確認するプログラム実行手段と、このプログラム実行手段により、関数のリターン処理においてこのガード変数が破壊されていることが検出された場合に、プログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行う異常終了実行手段とを備えることを特徴としている。このような構成とすれば、手作業を含むどのような手段でプログラムにガード変数を格納する指令を付加した場合であっても、そのようなプログラムを実行する際にスタック保護を実行できる点で好ましい。

【００２４】また、本発明は、種々の演算処理を行うデータ処理装置と、このデータ処理装置が演算処理を行う際に用いるメモリ装置とを備えたコンピュータシステムであって、このデータ処理装置が演算処理を実行する際の関数呼び出し後におけるこのメモリ装置のメモリパターンは、実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、このリターンアドレスを格納する領域とこのプレヴィアス・フレームポインタを格納する領域との後方に位置し、ローカル変数を格納する領域とを備え、このローカル変数を格納する領域は、このローカル変数を格納する領域に配列が格納されている場合に、この配列よりも前に、実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したことを特徴としている。このような構成とすれば、リターンアドレスと共にプレヴィアス・フレームポインタをも保護することにより、スタックスマッシング攻撃を用いたシステムへの侵入を確実に阻止することができる。さらに、ガード変数を用いたスタック保護処理の対象を、ローカル変数を格納する領域に配列が格納されている場合に限定することにより、システム全体のオーバーヘッドを低減させることができる。

【００２５】また、本発明は、種々の演算処理を行うデータ処理装置と、このデータ処理装置が演算処理を行う際に用いるメモリ装置とを備えたコンピュータシステムであって、このデータ処理装置が演算処理を実行する際の関数呼び出し後におけるこのメモリ装置のメモリパターンは、実行中の関数の呼び出し元へのリターンアドレスを格納する領域と、このリターンアドレスを格納する領域の後方に位置し、実行中の関数の呼び出し元へのプレヴィアス・フレームポインタを格納する領域と、このプレヴィアス・フレームポインタを格納する領域の後方に位置し、ローカル変数を格納する領域とを備え、このローカル変数を格納する領域に配列が格納されている場合に、このプレヴィアス・フレームポインタとこの配列との間に、実行中の関数のリターン処理において破壊されていないかどうかを確認する対象であるガード変数を格納したことを特徴としている。ガード変数を格納する位置は、スタック上のプレヴィアス・フレームポインタと配列との間であれば、どこでも良い。

【００２６】さらにまた、本発明は、プログラム制御により種々の演算処理を行うコンピュータシステムであって、関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令が付加されたプログラムを読み込んで実行するデータ処理装置と、このデータ処理装置が演算処理を行う際に用いるメモリ装置とを備え、このデータ処理装置は、前記ガード変数を格納する指令が付加されたプログラムにおけるガード変数を格納する指令にしたがって、関数呼び出しの際に、メモリ装置上のスタックにこのガード変数を格納すると共に、この関数のリターン処理においてこのガード変数の有効性を確認することを特徴としている。こ

のような構成とすれば、ガード変数を格納する指令が付加されたプログラムを、磁気ディスクや光ディスクなどの種々の記憶媒体に記録して提供したり、ネットワークを介して提供したりすることにより、このプログラムをインストールしたコンピュータシステムにおいて、スタック保護を実行できる点で好ましい。

【００２７】ここで、データ処理装置は、関数のリターン処理においてガード変数が破壊されていることを検出した場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行うことを特徴としている。このような構成とすれば、スタックスマッシング攻撃が行われた場合には、かかる関数の呼び出し元へ処理が戻る直前でプログラムの実行を中止するため、システムへの侵入を効果的に防止することができる点で好ましい。

【００２８】また、本発明は、プログラム制御により種々の演算処理を行うコンピュータシステムであって、ソース形式のプログラムを入力し、関数呼び出し後のスタックにおけるプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納する指令をこのソース形式のプログラムに付加すると共に、このガード変数を格納する指令を付加されたプログラムを実行するデータ処理装置と、このデータ処理装置が演算処理を行う際に用いるメモリ装置とを備え、このデータ処理装置は、このガード変数を格納する指令を付加されたプログラムにおけるガード変数を格納する指令にしたがって、関数呼び出しの際に、このメモリ装置上のスタックにこのガード変数を格納すると共に、この関数のリターン処理においてこのガード変数の有効性を確認することを特徴としている。このような構成とすれば、任意のプログラムに対して、ガード変数を格納する指令を付加し、このプログラムを実行することによって、スタックスマッシング攻撃を用いたシステムへの侵入を確実に阻止することができる点で優れている。ここで、プログラムは、コンパイラ言語で記述されたものであっても、インタプリタ言語で記述されたものであってもかまわない。コンパイラ言語で記述されたプログラムである場合は、データ処理装置は、コンパイラとしての機能も併せて有することとなる。

【００２９】ここで、データ処理装置は、関数のリターン処理においてガード変数が破壊されていることを検出した場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する異常終了処理を行うことを特徴としている。このような構成とすれば、スタックスマッシング攻撃が行われた場合には、かかる関数の呼び出し元へ処理が戻る直前でプログラムの実行を中止するため、システムへの侵入を効果的に防止することができる点で好ましい。

【００３０】また、本発明は、ソースプログラムを入力しオブジェクトプログラムに翻訳して出力するコンパイラであって、プログラムを翻訳する翻訳手段と、プログラムの翻訳の際に、サブルーチンごとに、このサブルーチンに含まれる関数が配列を持っているかどうかを調べ、配列を持っている場合に、このサブルーチンに対して、この関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納すると共に、関数のリターン処理においてこのガード変数の有効性を確認する旨の指令を付加するスタック保護指令付加手段とを備えることを特徴としている。このような構成とすれば、ソースプログラムをコンパイルした時点で必要な指令をプログラムに付加することができる点で優れている。

【００３１】さらにまた、本発明は、コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護方法であって、ソース形式のプログラムに対して、関数呼び出し後のスタックにおけるプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納する指令をこのソース形式のプログラムに付加するステップと、このガード変数を格納する指令を付加されたプログラムを実行すると共に、このガード変数を格納する指令にしたがって、関数呼び出しの際にスタックにこのガード変数を格納し、関数のリターン処理においてこのガード変数の有効性を確認するステップと、この関数のリターン処理においてこのガード変数が破壊されていることが検出された場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知するステップとを含むことを特徴としている。

【００３２】同様にして、コンピュータによるプログラムの実行の際にスタックスマッシング攻撃からプログラムカウンタを保護するスタック保護方法であって、関数を実行する際にスタック上におけるプレヴィアス・フレームポインタとローカル変数としてスタックに格納される配列との間にガード変数を格納する指令が付加されたプログラムを実行すると共に、このガード変数を格納する指令にしたがって、関数呼び出しの際にスタックにこのガード変数を格納し、関数のリターン処理においてこのガード変数の有効性を確認するステップと、関数のリターン処理においてこのガード変数が破壊されていることが検出された場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知するステップとを含む構成とすることができる。このような構成とすれば、手作業を含むどのような手段でプログラムにガード変数を格納する指令を付加した場合であっても、そのようなプログラムを実行する際にスタック保護を実行できる点で好ましい。

【００３３】また、本発明は、コンピュータに実行させるプログラムを当該コンピュータの入力手段が読取可能

に記憶した記憶媒体において、このプログラムは、配列を持っている関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納する処理と、関数のリターン処理においてこのガード変数の有効性を確認する処理と、関数のリターン処理においてこのガード変数が破壊されていることが検出された場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する処理とをこのコンピュータに実行させることを特徴としている。このような構成とすれば、このプログラムを実行する全てのシステムにおいて、スタックスマッシング攻撃からプログラムカウンタを保護できる点で好ましい。なお、このプログラムは、その種類において何ら制限はなく、コンパイラ言語であってもインタプリタ言語であっても良い。

【００３４】さらにまた、本発明は、コンピュータに、配列を持っている関数を実行する際のスタックにおいてプレヴィアス・フレームポインタとローカル変数としてこのスタックに格納される配列との間にガード変数を格納する処理と、関数のリターン処理においてこのガード変数の有効性を確認する処理と、関数のリターン処理においてこのガード変数が破壊されていることが検出された場合に、このプログラムの実行を中止し、スタックスマッシング攻撃があったことをユーザに通知する処理とを実行させるプログラムを記憶する記憶手段と、この記憶手段からこのプログラムを読み出してこのプログラムを送信する送信手段とを備えたことを特徴としている。このような構成とすれば、このプログラム伝送装置からこのプログラムをダウンロードしてインストール可能なあらゆるシステムにおいて、スタックスマッシング攻撃からプログラムカウンタを保護できる。

【００３５】

【発明の実施の形態】以下、添付図面に示す実施の形態に基づいてこの発明を詳細に説明する。まず、本発明の概要を説明する。本発明では、プログラムを実行する際のメモリパターン（記憶域の割当て）において、スタック上のプレヴィアス・フレームポインタ（Previous frame pointer：ＰＦＰ）とローカル変数領域における配列との間に、スタックスマッシング攻撃からプログラムカウンタを保護するための特別な値であるガード変数を挿入する。そして、関数リターン処理において、ガード変数の有効性を確認することにより、スタックスマッシング攻撃の有無を確認する。スタックスマッシング攻撃においては、ローカル変数への代入操作を行い、境界チェックが不完全な場合に、そのローカル変数より上位のスタック領域を破壊する。したがって、プレヴィアス・フレームポインタと配列との間にガード変数を挿入されたスタックに対して、スタックスマッシング攻撃が行われると、関数の出口でガード変数が破壊されていることが検出される。この時点でプログラムの実行を止めることにより、当該関数の呼び出し元へのリターンアドレス及びプレヴィアス・フレームポインタを保護することができ、当該システムへの侵入を阻止することができる。

【００３６】図１は、本実施の形態におけるスタック保護システムの全体構成を説明する図である。図１において、符号１１１はソース形式のプログラムである。符号１１２はスタック保護指令作成部であり、ソース形式のプログラム１１１における所定の箇所にガード変数を格納するための指令を作成してプログラムに付加する。符号１２１は、ガード変数を格納するための処理指令とガード変数の格納位置とを付加されたプログラムである。符号１３１はスタックであり、コンピュータのメモリ装置におけるメモリパターンを形成する。符号１３２はスタック保護処理実行部であり、プログラム１２１を実行すると共に、その際にプログラム１２１に付加されている処理指令及びガード変数の格納位置に基づいてスタックの保護処理を実行する。符号１３３は保護数値であり、ガード変数として用いられる値である。

【００３７】図２は、コンピュータにおいて、本実施の形態によりスタックの保護処理が実行されている場合におけるメモリ装置のスタックの状態（メモリパターン）を示す図である。図２に示すように、スタックトップ（図では下側）から順に、ローカル変数領域２１０、プレヴィアス・フレームポインタ（ＰＦＰ）２２０、及び当該呼び出し元へのリターンアドレス（ＲＡ）２３０が積まれている。そして、ローカル変数領域２１０における配列２１１の前、すなわち、配列２１１とプレヴィアス・フレームポインタ（ＰＦＰ）２２０との間にガード変数（Ｇ）２１２が格納されている。ガード変数（Ｇ）２１２の位置は、配列２１１とプレヴィアス・フレームポインタ（ＰＦＰ）２２０との間であればよく、この位置に格納される他のローカル変数との前後関係は問わない。

【００３８】図３は、図１に示したスタック保護システムにおけるスタック保護指令作成部１１２の処理を説明するフローチャートである。スタック保護指令作成部１１２は、ソース形式のプログラム１１１のサブルーチンごとに、図３に示す処理を実行する。図３を参照すると、スタック保護指令作成部１１２は、まず着目中のサブルーチンにおいて、ローカル変数として配列が使用されているかどうかを確認する（ステップ３０１）。そして、配列が使用されている場合は、ガード変数（Ｇ）の格納位置を決定する（ステップ３０２）。ここで、ガード変数（Ｇ）の格納位置は、上述したように、プレヴィアス・フレームポインタと配列との間の任意の位置である。次に、当該サブルーチンを実行する際に、スタックにおける、ステップ３０２で決定された当該ガード変数（Ｇ）の格納位置にガード変数（Ｇ）を格納する旨の処理指令を生成する（ステップ３０３）。以上のようにし

て、ガード変数を格納するための処理指令とガード変数の格納位置とを付加されたプログラム１２１が出力される。

【００３９】プログラム１２１には、実際には、ガード変数を挿入すべきサブルーチン（関数）ごとに、ガード変数の宣言、関数の入口におけるガード変数の値の設定、関数の出口におけるガード変数の値の確認及び異常終了を実行するための命令が埋め込まれることとなる。このプログラム１２１の具体的な構成例は後述する。

【００４０】図４は、スタック保護処理実行部１３２の処理を説明するフローチャートである。スタック保護処理実行部１３２は、スタック保護指令作成部１１２によりガード変数を格納するための処理指令とガード変数の格納位置とを付加されたプログラム１２１のサブルーチンごとに、図４に示す処理を実行する。図４を参照すると、まずプログラム１２１の着目中のサブルーチンにおいて、スタック保護指令作成部１１２により付加された処理指令があるかどうかを調べる（ステップ４０１）。そして、処理指令がある場合は、当該プログラム１２１に処理指令と共に付加されているガード変数（Ｇ）格納位置に対応するスタック１３１上の位置にガード変数（Ｇ）の格納領域を生成する（ステップ４０２）。上述したように、ガード変数（Ｇ）の格納領域は、スタック１３１上のローカル変数領域であって、配列よりも前の位置に生成される。

【００４１】次に、ステップ４０２で生成されたガード変数（Ｇ）の格納領域に、保護数値１３３を格納する（ステップ４０３）。これにより、スタック１３１は、図２に示した状態となる。ガード変数（Ｇ）として格納される保護数値１３３の内容は、特に制約はないが、攻撃者が推定できない値や攻撃によって格納することのできない値とする。そのような値の例として乱数を利用することができる。

【００４２】次に、当該サブルーチンの処理（関数）を実行する（ステップ４０４）。そして、関数の出口でガード変数（Ｇ）として格納した値が保持されているかどうかを確認する（ステップ４０５）。ガード変数（Ｇ）の値が保持されていれば、通常通り終了する。また、ガード変数（Ｇ）の値が保持されていない場合は、スタックスマッシング攻撃が行われたことを意味するので、攻撃があったことを示すメッセージ及びエラーログを出力してプログラムを終了する（ステップ４０６）。以上のようにして、スタックスマッシング攻撃があれば、リターン処理の直前で検出でき、コンピュータシステムへの侵入を防止できることとなる。

【００４３】本実施の形態のスタック保護システムは、結果的にプログラムの実行段階において、メモリパターンとして、図２に示したように、スタック上のプレヴィアス・フレームポインタと配列との間にガード変数を配置できれば良い。したがって、本実施の形態によるスタック保護システムを実現するにあたり、プログラミング言語の種類やシステムの形態に応じて、種々の態様を採ることができる。例えば、プログラミング言語がコンパイラ言語である場合は、スタック保護指令作成部１１２の機能をコンパイラに実装することができる。この場合、スタック保護指令作成部１１２により処理指令及びガード変数（Ｇ）の格納位置を付加されたプログラム１２１は、実行形式のプログラムとなる。コンパイラのコード生成時にガード変数を挿入するためのプログラムを埋め込むこととすれば、オーバーフローを引き起こす変数よりもスタック上で上位アドレスにガード変数の宣言を配置すること、及び全ての関数の出口にプログラムを挿入することが容易となる点で好ましい。

【００４４】これに対し、プログラミング言語がインタプリタ言語である場合は、スタック保護指令作成部１１２により処理指令及びガード変数（Ｇ）の格納位置を付加されたプログラム１２１の段階では、未だソース形式のプログラムである。また、プログラミング言語がコンパイラ言語である場合であっても、コンパイラの前段にスタック保護指令作成部１１２を構成する処理装置を置き、ソースプログラムの段階でガード変数を挿入するためのプログラムを埋め込むようにしても良い。図１０に示したＣ言語のソースプログラムに対してガード変数を挿入するためのプログラムを埋め込む例を図５及び図６を参照して説明する。この場合、スタック保護指令作成部１１２は、図５に示す３つのプログラムを、図１０のプログラムの変数宣言部、関数の入口、関数の出口にそれぞれ挿入する。各プログラムが挿入された状態が図６に示すプログラムである。図６を参照すると、変数宣言部である符号６０１の位置、関数の入口である符号６０２の位置、関数の出口である符号６０３の位置に、それぞれ図５に示した３つのプログラムが挿入されている。

【００４５】このように、スタック保護システムを実現するには、プログラミング言語の種類やシステムの形態に応じて、種々の態様を採ることが可能である。上記の他にも、ソース形式のプログラムを作成する段階で、所望の関数に対して、手作業により図５に示したようなプログラムを書き込むことも可能である。この場合、スタック保護指令作成部１１２の処理が手作業にて行われることに相当する。したがって、作成されたプログラムは最初からプログラム１２１の状態であり、スタック保護システムとしては、スタック保護処理実行部１３２による処理だけが実行されることとなる。

【００４６】図７は、本実施の形態のスタック保護システムを適用したコンピュータシステムの構成例を説明する図である。図７に示すコンピュータシステム７００は、データ処理を行うデータ処理装置７１０と、データ処理装置７１０による処理の際に用いられるメモリ装置７２０と、データ処理装置７１０を制御する制御プログラム７３１を格納した記憶装置７３０とを備える。ＣＰ

Ｕにて実現されるデータ処理装置７１０は、制御プログラム７３１の制御の下、スタック保護処理実行部１３２の機能を実現する。メモリ装置７２０は、スタック保護処理実行部１３２による関数の実行において、スタック７２１上に、図２に示したガード変数を含むメモリパターンを形成する。さらに、データ処理装置７１０のスタック保護処理実行部１３２は、メモリ装置７２０のスタック７２１上に図２に示したメモリパターンを形成する際に、ガード変数の値として、記憶装置７３０から読み出した保護数値１３３を挿入する。なお、コンピュータシステム７００において、コンパイラ言語が実行される場合は、制御プログラム７３１は、図１に示したプログラム１２１に相当する実行形式のプログラムとなる。また、記憶装置７３０に格納された制御プログラム７３１を、ガード変数を格納するための処理指令とガード変数の格納位置とを付加されたソース形式のプログラムとすることもできる。この場合、データ処理装置７１０は、コンパイラの機能を併せて備えることとなる。さらに、制御プログラム７３１を、ガード変数を格納するための処理指令とガード変数の格納位置とを付加する前のソース形式のプログラムとすることもできる。この場合、制御プログラム７３１は、図１に示したソース形式のプログラム１１１に相当し、データ処理装置７１０は、スタック保護指令作成部１１２の機能を更に備えることとなる。さらにまた、図７においては、メモリ装置７２０と記憶装置７３０とを別個の構成要素として記載したが、物理的には単一の記憶モジュールにて構成することができる。

【００４７】次に、本実施の形態におけるオーバーフローの低減について説明する。変数の型宣言とその参照とが正しく対応しているプログラムにおいては、バッファオーバーフローが発生する変数は、サイズ情報を持たない配列である。Ｃ言語などで使用されるこのような配列は、サイズ情報がないため、コンパイラの型チェックだけではバッファオーバーフローを防ぐことができない。一方、配列以外の変数では、プログラミング作法として型変換に注意を払えば、バッファオーバーフローを防ぐことが可能である。したがって、上述したように本実施の形態では、図２に示したように、スタックのローカル変数領域における配列とプレヴィアス・フレームポインタとの間にガード変数を格納し、ローカル変数として配列を持たない場合はガード変数を格納しないこととした。これにより、システム全体のオーバーヘッドの低減を図ることができる。

【００４８】ここで、変数の型宣言とその参照とが正しく対応しているプログラムにおいて、テスト領域に配列を持っていない関数（ＡＦ関数と称す）では、ガード変数を用いたバッファオーバーフローのテストを行う必要がないことを証明する。図８は、スタックにおけるテスト領域を説明する図である。スタック上のガード変数を用いてバッファオーバーフローのテストを行う領域は、当該ガード変数とそれよりも下位に現れるガード変数との間の領域、すなわち、図８の斜線を付した領域である。この領域を関数のテスト領域８０１と呼ぶこととする。証明は次の２つのステップで行う。１．ＡＦ関数のテスト領域８０１が破壊されるのは、それ以降に呼ばれる関数のテスト領域８０１がバッファオーバーフローを起こした場合であり、またその場合だけであることを示す。２．ＡＦ関数のテスト領域８０１が破壊された場合は、ＡＦ関数に実行制御が戻らないことを示す。

【００４９】まず、第１のステップについて説明する。ＡＦ関数のテスト領域８０１には配列は存在しない。したがって、テスト領域８０１が破壊されたとしても、当該テスト領域がバッファオーバーフローの発生源ではない。したがって、テスト領域８０１の破壊の原因は、スタック上の当該関数フレームの上に置かれたテスト領域８０１となる。すなわち、ＡＦ関数から呼び出された関数の中でバッファオーバーフローが生じ、スタックの上位アドレスの向きに破壊して、ＡＦ関数のテスト領域を壊したものである。

【００５０】次に、第２のステップについて説明する。第１のステップにより、ＡＦ関数のテスト領域８０１の破壊原因は、それ以降に呼ばれる他の関数にある。また上述したように、変数の型宣言とその参照とが正しく対応しているプログラムにおいては、バッファオーバーフローが発生する変数は配列であるから、ＡＦ関数におけるテスト領域８０１の破壊の原因となった領域は、配列である。配列を定義した関数では、ガード変数を用いたバッファオーバーフローのテストが実行されている。そして、テスト領域８０１を破壊し、かつリターンアドレスを操作するような攻撃があった場合は、必ずガード変数が破壊される。そのため、当該関数のリターン処理でプログラムは停止する。したがって、ＡＦ関数まで実行制御が戻ることはない。

【００５１】以上で、テスト領域８０１に配列を持っていないＡＦ関数では、ガード変数を用いたバッファオーバーフローのテストを行う必要がないことが証明された。なお、本実施の形態では、配列一般に対してガード変数を用いたスタック保護を行うこととして説明したが、特に文字配列のみを対象としてガード変数を用いたスタック保護を行うこととしても良い。これは、スタックスマッシング攻撃の対象となる配列が主に文字配列だからである。さらにその理由は、文字配列が、プログラム外部の情報表現に使用すること、データサイズを確認せずに処理できるように終端記号を持つ表現であること、といった特徴を持つことにある。したがって、ガード変数を用いたスタック保護を行う関数を、ローカル変数として文字配列を持つものに限定し、文字配列を持たないもの及び整数配列のような文字配列以外の配列を持つものについては、ガード変数を格納しないことによ

り、システム全体のオーバーヘッドを更に低減させることができる。

【００５２】以上のように、本実施の形態では、関数のリターン処理において、スタックのプレヴィアス・フレームポインタと配列との間に格納されたガード変数の有効性を確認する。これにより、スタック上のバッファオーバーフローを原因とするスタックスマッシング攻撃に関しては、ネットワーク経由での攻撃及びローカルユーザからの攻撃のいずれに対しても防止することができる。

【００５３】また、本実施の形態では、ローカル変数として配列を持つ関数のみを対象としてガード変数を用いたスタック保護を実行するため、従来技術において説明したように、全ての関数に対してスタック保護のための処理を実行する場合と比べてオーバーヘッドの削減を図ることができる。特にリターンアドレスの格納箇所を書き込み禁止にする場合や、リターンアドレス専用のスタックを用意する場合と比べると、大幅なオーバーヘッドの削減を実現できる。さらに、ローカル変数として文字配列を持つ関数のみを対象としてガード変数を用いたスタック保護を行うこととすれば、さらなるオーバーヘッドの削減を図ることができる。

【００５４】なお、本実施の形態では、上述したようにガード変数をローカル変数領域に格納するため、スタックのフレーム構造を変更することがない。したがって、プログラムに対するデバッガによるデバッグ支援処理を行うことができなくなると行った不都合もない。

【００５５】

【発明の効果】以上説明したように、本発明によれば、スタック上のバッファオーバーフローを原因とするスタックスマッシング攻撃を完全に防止することができる。また、スタックスマッシング攻撃の防止手段を実装したことによる実行オーバーヘッドを削減することができる。

【図面の簡単な説明】

【図１】　本実施の形態におけるスタック保護システムの全体構成を説明するための図である。

【図２】　本実施の形態によりスタックの保護処理が実行されている場合のスタックの状態（メモリパターン）を示す図である。

【図３】　本実施の形態におけるスタック保護指令作成部の処理を説明するフローチャートである。

【図４】　本実施の形態におけるスタック保護処理実行部の処理を説明するフローチャートである。

【図５】　本実施の形態においてスタック保護の対象とすべきプログラムに挿入する、スタック保護を実行するためのプログラムの例を示す図である。

【図６】　図１０のプログラムに対して図５のプログラムを挿入した状態を示す図である。

【図７】　本実施の形態のスタック保護システムを適用したコンピュータシステムの構成例を説明する図である。

【図８】　スタックにおけるテスト領域を説明する図である。

【図９】　Ｃ言語における関数呼び出し後のスタック状態を説明する図である。

【図１０】　Ｃ言語で記述されたプログラムの例を示す図である。

【図１１】　図１０のプログラムに対してスタックスマッシング攻撃が行われた場合のスタックの状態を説明する図である。

【符号の説明】

１１１…ソース形式のプログラム、１１２…スタック保護指令作成部、１２１…プログラム、１３１…スタック、１３２…スタック保護処理実行部、１３３…保護数値、７００…コンピュータシステム、７１０…データ処理装置、７２０…メモリ装置、７２１…スタック、７３０…記憶装置、７３１…制御プログラム、８０１…テスト領域

【図５】

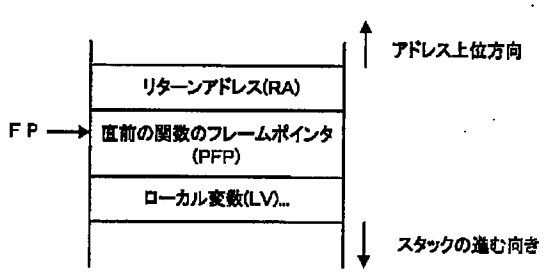
● 変数宣言
```
volatile int  guard;
```
● 関数入口
```
gv = guard_value;
```
● 関数出口
```
if (gv!= guard_value){
    /*output error log */
    /*halt execution */
}
```

【図６】

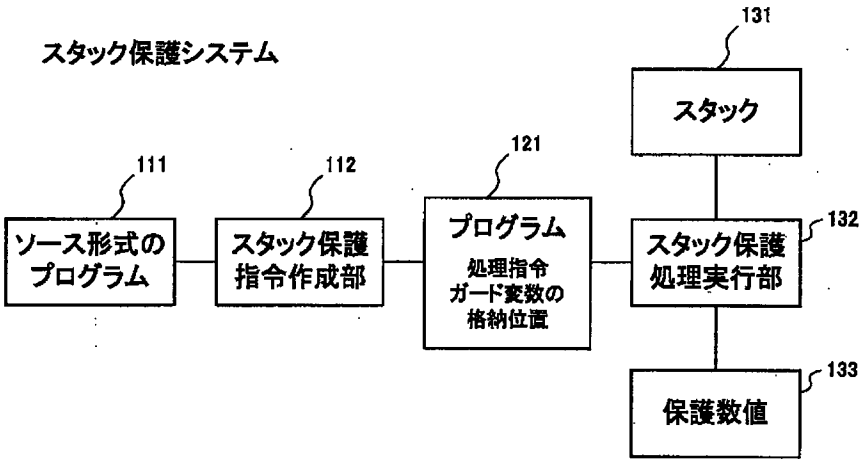
```
void foo()
{
        volatile int guard;     ←― 601
        char buf[128];

        gv = guard_value;       ←― 602
        ----
        strcpy (buf, getenv ("HOME"));
        ----
        if (gv!= guard_value){
            /*output error log */     ←― 603
            /*halt execution */
        }
}
```

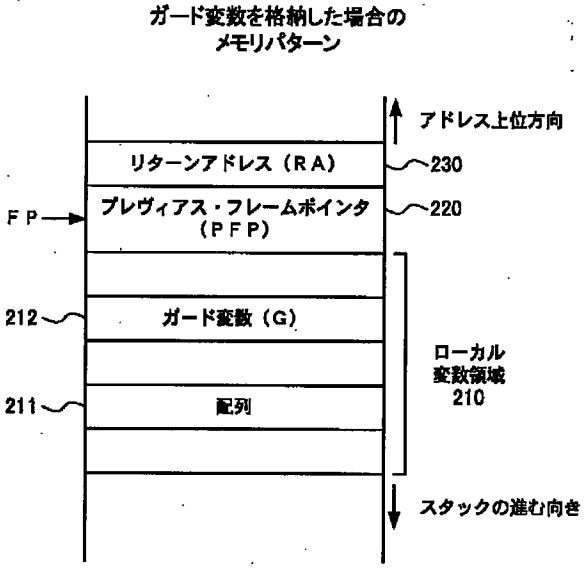
【図９】



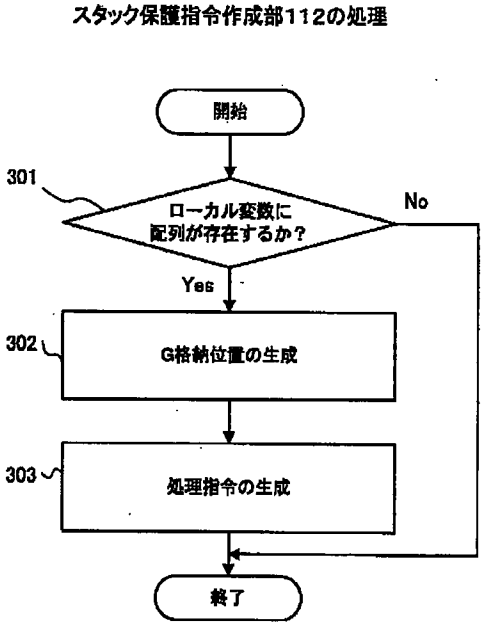
【図１】



【図２】



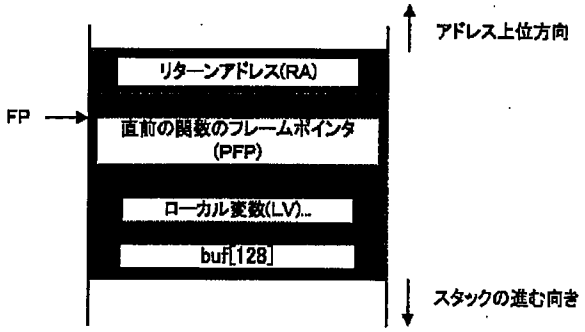
【図３】



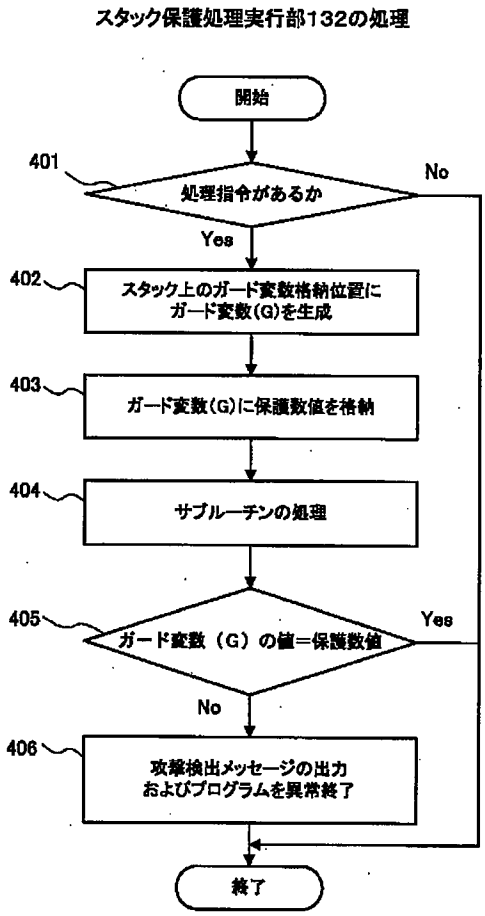
【図１０】

```
void foo()
{
        char buf[128];
        ----
        strcpy (buf, getenv ("HOME"));
        ----
}
```

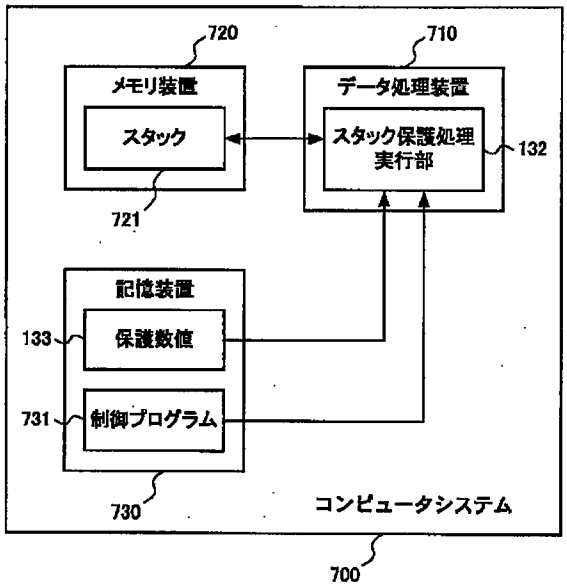
【図１１】



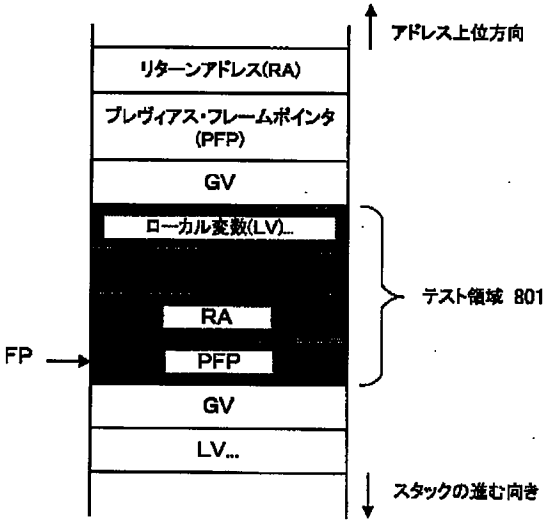
【図４】


スタック保護処理実行部132の処理
開始
401
処理指令があるか
No
Yes
402
スタック上のガード変数格納位置に
ガード変数(G)を生成
403
ガード変数(G)に保護数値を格納
404
サブルーチンの処理
405
ガード変数（G）の値＝保護数値
Yes
No
406
攻撃検出メッセージの出力
およびプログラムを異常終了
終了

【図７】


720
710
メモリ装置
スタック
データ処理装置
スタック保護処理
実行部
132
721
記憶装置
133
保護数値
731
制御プログラム
730
コンピュータシステム
700

【図８】


アドレス上位方向
リターンアドレス(RA)
プレヴィアス・フレームポインタ
(PFP)
GV
ローカル変数(LV)...
RA
PFP
FP
テスト領域 801
GV
LV...
スタックの進む向き

フロントページの続き

(72)発明者　江藤　博明
神奈川県大和市下鶴間1623番地14　日本アイ・ビー・エム株式会社　東京基礎研究所内

(72)発明者　依田　邦和
神奈川県大和市下鶴間1623番地14　日本アイ・ビー・エム株式会社　東京基礎研究所内

Fターム(参考)　5B033 AA10 DE05 DE07 FA01 FA27